The WebdamLog System: Managing Distributed Knowledge on the Web*

Serge Abiteboul [1,2], Émilien Antoine [2] and Julia Stoyanovich [3]
serge.abiteboul@inria.fr emilien.antoine@inria.fr jstoy@seas.upenn.edu
[1]Collège de France, Paris, France [2]Inria Saclay & ENS Cachan, France [3]University of Pennsylvania, USA

Résumé

Ce papier traite de la gestion des bases de connaissances distribuées, dans des environnements où un grand nombres de pairs autonomes et hétérogènes collaborent pour réaliser des tâches. Les informations sont représentées par des faits et les tâches correspondent à des règles logiques. Nous utilisons le langage *WebdamLog*, une variante de datalog pour les données distribuées. Nous présentons l'implémentation d'un moteur *WebdamLog* et nous montrons son efficacité dans le cadre d'applications distribuées dans lesquels les pairs changent rapidement leurs base de connaissances. Notre implémentation s'appuie sur *Bud* pour l'évaluation standard de datalog. Le moteur *WebdamLog* étends *Bud* avec le support (1) de règles avec des prédicats non locaux dans le corps ; (2) de règles avec des variables pour les prédicats ou les noms des pairs ; (3) la gestion des délégations qui changent le programme des pairs distants.

Keywords: Datalog, Distributed, Declarative, Knowledge base

1 Introduction

The need for a unified approach for the management of "distributed content" has been argued in a number of works such as *P2P Content Warehouse* [1], *Dataspaces* [10] and *Data rings* [5]. Query and update facilities in a heterogeneous distributed environment consisting of autonomous peers form the backbone of such systems. Personal information management is often given as an important motivating example [10]. In [5], we argued for the use of declarative languages to facilitate the adoption of our platform by non-technical users, for the purposes of data exchange.

In this work, we build on the insights of [5] and [10] and propose to use *WebdamLog* [4], a declarative high-level language in the style of datalog, to support the distribution of both *data* and *knowledge* (i.e., programs) over a network of heterogeneous peers.

Datalog, a prehistoric language by Web time, includes recursion, an essential aspect for managing graphs, notably the Web graph. In recent years, there has been renewed interest in applying languages in the datalog family for a range of applications, from program analysis, to security and privacy protocols, to natural language processing, to multi-player games. The arguments in

*This work has been partially funded by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013); ERC grantWebdam, agreement 226513. http://webdam.inria.fr/

favor of datalog-style languages are familiar ones: a declarative approach alleviates the conceptual complexity on the user, while at the same time allowing for powerful performance optimizations.

WebdamLog is a datalog-style language that emphasizes cooperation between distributed autonomous peers communicating in an asynchronous manner, and is well adapted to the evolution of the peers' knowledge over time. As we will see in Section 3, *WebdamLog* extends datalog in a number of ways, supporting updates [6, 7], distribution [2], negation [11], and, importantly, a novel feature called delegation [4]. As a result, *WebdamLog* is neither as simple nor as beautiful as datalog. It is also more procedural, which is needed to capture real Web applications.

The rest of this paper is organized as follows. In Section 2, we illustrate the salient features of *WebdamLog* using an example motivated by personal information management. We give an overview of *WebdamLog* in Section 3, describe our implementation of the engine in Section 4, and present results of a preliminary experimental evaluation in Section 5. We conclude in Section 6.

2 Running example

Suppose that Alice and Bob are getting married, and their friends want to offer them an album of photos in which the bride and groom appear together. Such photos may be owned by friends and relatives of Alice and Bob. Owners of the photos may store them on a variety of services and devices, including, e.g., desktop computers, smartphones, Picasa, and Flickr.

Making a photo album for Alice and Bob involves the following steps: (1) Identify friends and relatives of Alice and Bob using Facebook and Google+; (2) Find out where they keep their photos and how to access these photos; (3) From among all photos, select those that feature both Alice and Bob, using, e.g., tags or face recognition software; (4) Ask Alice's friend Sue to verify that the gathered and filtered photos are appropriate for the photo album, which will be seen by the families of Alice and Bob on their wedding day; (5) Sue may realize that inappropriate photos of the couple come from Dave's Flickr stream, and may decide to exclude Dave from the set of sources.

As should be apparent from this example, a task of this kind would be much more manageable if it were executed automatically. Executing such a task involves a certain amount of reasoning on the part of the system, which can be encoded with declarative rules. For instance, the following rule computes the union of Alice's and Bob's Facebook contacts in relation *source* on Sue's peer:

```
[rule at sue] source@sue($name) :- friends@aliceFB($name)
              source@sue($name) :- friends@bobFB($name)
```

using wrappers to Facebook for Alice and Bob.

For simplicity, we assume that a friend's name corresponds to the name of the *peer* that the particular friend uses as entry point to the system. This name is thus associated to a particular URL. We assume that, in particular, each peer keeps the localization data of the friend, e.g., where the friend keeps her photos. The following rule, at peer *sue*, *delegates* steps (2) and (3) to the peers corresponding to the friends of Alice and Bob:

```
[rule at sue] album@sue($photo,$name) :- source@sue($name),
                        photoLocation@$name($peer),
                        photos@$peer($photo),
                        features@$peer($photo,alice),
                        features@$peer($photo,bob)
```

The key feature of this rule is the use of the *WebdamLog* language to share the work. Let Dan be a friend, and so a possible source. Then Sue's peer will delegate the following rule to Dan's peer:

```
[rule at dan] album@sue($photo,dan) :- photoLocation@dan($peer),
                        photos@$peer($photo),
                        features@$peer($photo,alice),
                        features@$peer($photo,bob)
```

Now suppose that Dan uses both Picasa and Flickr. Then, Dan's peer will delegate to *danPicasa*, a wrapper for his Picasa account, the following rule (and a similar rule for Flick):

```
[rule at danPicasa] album@sue($photo,dan) :- photos@danPicasa($photo),
                              features@danPicasa($photo,alice),
                              features@danPicasa($photo,bob)
```

Note how the tasks are automatically shared by many peers. Observe that when new friends of Alice or Bob are discovered (e.g., proposed by some known friends), Sue's album, which is defined *intensionally*, will be automatically updated. Observe also that, to simplify, we assume here that all peers use a similar organization (ontolog y). This constraint may easily be removed at the cost of slightly more complicated rules.

To allow for manual curation by Sue, it suffices to modify the definition of sources:

```
[rule at sue] source@sue($name) :- friends@aliceFB($name), not blocked@sue($name)
              source@sue($name) :- friends@bobFB($name), not blocked@sue($name)
```

By inserting/removing facts in *blocked@sue*, Sue now controls who can participate. (A similar control can also be added at the photo level.) Observe that such updates result in modifying the programs running at the participating peers. The album evolves, controlled by Sue's updates as well as by the discovery of new friends of Alice or Bob, and of new sources of photos.We will use the example of this section throughout the paper to demonstrate the salient features of our approach.

3 Webdamlog Model

In this section, we briefly recall the semantics of *WebdamLog* from [4]. We assume the existence of a countable set of data values (that includes a set of relation names and a set of peer names) and of a countable set of variables. Variables start with the symbol \$, e.g. $\$x$.

Schema A *relation* in our context is an expression $m@p$ where m is a relation name and p a peer name. A *schema* is an expression (π, E, I, σ) where π is a possibly infinite set of peer names, E is a set of extensional relations of the form $m@p$ for $p \in \pi$, I is a set of intensional relations of the form $m@p$ for $p \in \pi$, and σ, the sorting function, specifies for each relation $m@p$, an integer $\sigma(m@p)$ that is its sort. A relation cannot be at the same time intensional and extensional.

Facts A *fact* is an expression of the form $m@p(a_1, ..., a_n)$, where $n = \sigma(m@p)$ and $a_1, ..., a_n$ are data values. An example of a fact is: $\mathsf{pictures@myalbum}(1771.jpg, "Paris", 11/11/2011)$

Rules A *term* is a constant or a variable. A rule in a peer p is an expression of the form:
[at p] $\$\mathsf{R}@\$\mathsf{P}(\$\mathsf{U})$:- $(\neg)$ $\$\mathsf{R}_1@\$\mathsf{P}_1(\$\mathsf{U}_1)$,..., $(\neg)$ $\$\mathsf{R}_n@\$\mathsf{P}_n(\$\mathsf{U}_n)$

Here, R, R_i are relation terms, P, P_i are peer terms, U, U_i are tuples of terms. We impose the safety condition that $R and $P must appear positively bound in the body and that each variable occurring in a negative literal must also appear positively bound in the body. We evaluate rules from left to right and require that any peer name P_i$ must be bound in a previous atom, i.e., we disallow broadcast.

Semantics Each peer p has a *state* consisting of some facts, some rules, and possibly of some rules that are delegated by other peers. Peers evolve by updating their base of facts, by sending facts to other peers, and by updating their delegations to other peers. So, both the set of facts and the set of delegated rules evolve over time. (To simplify, we assume that the set of rules specified locally is fixed.)

The semantics of a rule with head m@p(u) in a peer p′ depends on the nature of the relation in its head: whether it is extensional (m@p in E) or intensional (m@p in I), and whether it is local (p=p′) or not. We first consider rules in which all relations occurring in the body are local; we call such rules *local rules*. A subtlety lies in the use of variables for peer names. The nature of a rule may depend on the instantiation of these variables. For example, one instantiation of a particular rule may be local, whereas another may not be.

Local with local intensional head (datalog) These rules define local intensional predicates, as in classic datalog.

Local with local extensional head (local database updates) Facts derived by this kind of a rule are inserted into the local database. Note that, by default, like in Dedalus, facts are not persistent. To have them persist, we use rules of the form m@p(U) :- m@p(U). Deletion can be captured by controlling the persistence of facts.

The two kinds of rules described above, containing only predicates of the local peer, are called *fully local rules*. These rules do not require network communication, and are not affected by problems due to asynchronicity of the network.

Local with non-local extensional head (messaging) Facts derived by rules of this kind are sent to other peers. For example, the rule:

[at mi] $m@$p($name, "*Happybirthday!*") :- today@mi($date),
birthday@mi($name, $m, $p, $date)

where mi stands for *my iPhone*, results in sending a Happy Birthday message to a contact on the day of his birthday. Observe that the name $p of the peer and the name $m of the message varies depending on the person.

Local with non-local intensional head (view delegation) Such a rule results in installing a view remotely. For instance, the rule

[at mi] boyMeetsGirl@gossipsite($girl, $boy) :- girls@mi($girl, $loc), boys@mi($boy, $loc)

installs a join of two mi relations at gossipsite.

Non-local (general delegation) Consider the rule

[at mi] boyMeetsGirl@gossipsite($girl, $boy) :- girls@mi($girl, $loc), boys@ai($boy, $loc)

where ai stands for Alice's iPhone. This results in installing, at gossipsite, a view t$_{r@mi}$ and a rule, defined as follows:

[at mi] t$_{r@mi}$@ai($girl, $loc) :- girls@mi($girl$loc)

[at ai] boyMeetsGirl@gossipsite($girl, $boy) :- $t_{r@mi}$@ai($girl, $loc), boys@ai($boy, $loc)

Note that both rules are now local. Note also that, when girls@mi changes, this modifies the view at Alice's iPhone, possibly changing the semantics of boyMeetsGirl@gossipsite.

In [4], we formally define the semantics of Webdamlog. We show that, unless all peers and programs are known in advance, delegation strictly increases the expressive power of the model. If they are known in advance, delegation does not bring any extra power. Of course, delegation is also useful in practice, because it enables obtaining logic (rules) from other sites, and deploying logic (rules) to other sites. Conditions for systems to converge are shown in [4], and are extremely restrictive. Even in the absence of negation, a *WebdamLog* system will typically not converge because of asynchronicity.

4 The *WebdamLog* System

In this section, we describe the architecture of the *WebdamLog* system. We describe the implementation of the system, stressing the novel features compared to standard datalog engines.

System architecture The *WebdamLog* language is motivated by previous works on the *WebdamExchange* system [3]. There, we described a system we built that could automatically adapt to a variety of protocols and access methods found on the Web, notably for localizing data and for access control [8]. In developing toy applications with [8], we realized the need for a logic that could be used (i) to *declaratively* specify applications and (ii) to exchange application logic between peers. This motivated the introduction of *WebdamLog*, a language based on rules that can run locally and that can also be exchanged between peers.

Datalog evaluation has been intensively studied, and several open-source implementations are available. We chose not to implement yet another datalog engine, but instead to extend an existing one. We use the *Bud* [9] system because of its support for asynchronous communication, and because its scalability has been demonstrated in real-life scenarios such as Internet routing.

WebdamLog* simulation in *Bud The *Bud* system supports a powerful datalog-like language introduced in [16]. Indeed, we see *Bud* (and use it) as a distributed datalog engine with updates and asynchronous communications.

A *WebdamLog* computation consists of a sequence of *stages*, with each stage involving a single peer. Each stage of a *WebdamLog* peer is in turn simulated by a three-step *Bud* computation, described next. Note that we use the word *stage* for *WebdamLog* and *step* for *Bud*. Step 1, inputs are collected from the external world, including input messages from other peers, clock interrupts and host language calls. Step 2, the time is frozen; the union of the local store and of the batch of events received since the last stage is taken as an extensional database, and the program is run to fixpoint. Step 3, outputs are obtained as side effects of the program, including output messages to other peers, updates to the local store, and host language callbacks.

Observe that the fixpoint computation is executed at Step 2. It is performed by a local datalog engine over a fixed set of extensional relations and a fixed set of rules with no deletion. Deletion messages may be produced at Step 3, along with updates to the set of rules (for different reasons that we will explain further), and so this occurs outside of the datalog fixpoint computation. Relations appearing in the rules are implemented as *Bud* collections. *Bud* distinguishes between

three kinds of key-value sets: A *table* keeps a fact until an explicit delete order is received. We use tables to support *WebdamLog* extensional relations. A *scratch* is used for storing results of intermediate computation, and is emptied at the beginning of Step 1. We use *scratch* collections to implement *WebdamLog* intensional relations. A *channel* provides support for asynchronous communication. It records facts that have to be sent to other peers, and messages related to installing or removing delegations.

Since communication is asynchronous, there is no guarantee as to when a fact written to a channel will be received by a remote peer. This is a departure from "pure" *WebdamLog* that essentially captures asynchronicity by considering only that the peers are fired in an arbitrary order. It is shown in [4] that communication delays can easily be captured in *WebdamLog* by introducing peers modeling the network. Also, in *WebdamLog*, facts in a peer are consumed by the engine at each firing of the peer. To make facts persistent, they have to be re-derived by the peer at each stage. This is captured in our implementation by assuming that rules re-derive extensional facts implicitly, unless a deletion message has been received. A subtlety is that deletions local to a peer are also sent to the local peer itself using a channel. For this reason, it is a bit tricky to guarantee that local deletion messages are received by the peer before the next stage is started.

Implementing *WebdamLog* rules We now describe how *WebdamLog* rules are implemented on top of *Bud*. We distinguish between 4 cases. As for the semantics of *WebdamLog* (see Section 3), whether a rule in a peer p is *local* (i.e., all relations occurring in rule body are p-relations) plays an important role. The last case focuses on the use of variables for relation and peer names. For the first 3 cases, we thus ignore such variables.

Basic case. This is the case with local rules with either an extensional relation or a local intensional relation in the head. Such *WebdamLog* rules are implemented by identical *Bud* rules. In the case of an extensional head, this takes care of messages for local updates and messages to other peers, and in the case of a local intensional head, of local deduction as in datalog.

Local with non-local intensional head. From an implementation viewpoint, this case is somewhat tricky. We illustrate it with an example. Consider an intensional relation $\mathsf{s_0@q}$ defined in the distributed setting by the following two rules:

[at p1] $\mathsf{s_0@q}$(X,Y):- $\mathsf{r_1@p_1}$(X,Y) [at p2] $\mathsf{s_0@q}$(X,Y):-$\mathsf{r_1@p_2}$(X,Y)

Intuitively, the two rules specify a view relation $\mathsf{s_0@q}$ at q that is the union of two relations $\mathsf{r_1@p_1}$ and $\mathsf{r_1@p_2}$ from peers $\mathsf{p_1}$ and $\mathsf{p_2}$, respectively. However, *Bud* does not support views. A naive implementation would materialize relation s_0 at q, and would then have $\mathsf{p_1}$ and $\mathsf{p_2}$ send update messages to q. Now suppose that a tuple $\langle 0, 1\rangle$ is in both $\mathsf{r_1@p_1}$ and $\mathsf{r_1@p_2}$. Then it is correctly in $\mathsf{s_0@q}$. Now suppose that this tuple is deleted from $\mathsf{r_1@p_1}$. Then a deletion message is sent to q, resulting in wrongly deleting the fact from $\mathsf{s_0@q}$.

The problem arises because the tuple $\langle 0, 1\rangle$ originally had two reasons to be in s_0, and only one of the reasons disappeared. To avoid this problem, we should record the *provenance* of the fact $\langle 0, 1\rangle$ in $\mathsf{s_0@q}$. For now, we can implement the following *Bud* rules at $\mathsf{p_1}$, $\mathsf{p_2}$ to fix the problem:

[at p1] $\mathsf{s_{0p1}@q}$(X,Y):- $\mathsf{r_1@p_1}$(X,Y) [at p2] $\mathsf{s_{0p2}@q}$(X,Y):- $\mathsf{r_1@p_2}$(X,Y)

allowing to join these two relations locally with

[at s] $\mathsf{s_0@q}$(X,Y):- $\mathsf{s_{0p1}@q}$(X,Y) [at s] $\mathsf{s_0@q}$(X,Y):- $\mathsf{s_{0p2}@q}$(X,Y)

Note that relations $\mathsf{s_{0p1}}$ and $\mathsf{s_{0p2}}$ may be either intensional, in which case the view is computed on demand, or extensional, in which case the view is materialized.

Non-local rules. We now consider non-local rules with extensional head. (Non-local rules with intensional head are treated similarly.) An example of such a rule is:

[at p] $\quad \mathsf{r_0@q}(\overline{X_0})\text{:-}\ \mathsf{r_1@q_1}(\overline{X_1}),\ldots,\mathsf{r_i@q_i}(\overline{X_i}),\ldots$

with $\mathsf{q_1} = \ldots = \mathsf{q_{i-1}} = \mathsf{p}$, $\mathsf{q_i} = \mathsf{q} \neq \mathsf{p}$, and with each $\overline{X_j}$ denoting a tuple of terms. If we consider atoms in the body from left to right, we can process at p the rule until we reach $\mathsf{r_i@q}(\overline{X_i})$. Peer p does not know how to evaluate this atom, but it knows that the atom is in the realm of q. Therefore, p rewrites the rule into two rules, as per definition of delegation in *WebdamLog*:

[at p] $\mathsf{mid@q}(\overline{X_{mid}})$:- $\mathsf{r_1@p}(\overline{X_1}),\ldots,\mathsf{r_{i-1}@p}(\overline{X_{i-1}})$
[at q] $\mathsf{r_0@q}(\overline{X_0})$:- $\mathsf{mid@q}(\overline{X_{mid}})$, $\mathsf{r_i@q}(\overline{X_i}),\ldots$

where *mid* identifies the message, and notably encodes, (i) the identifier of the original rule, (ii) that the rule was delegated by p to q, and (iii) the split position in the original rule. The tuple $\overline{X_{mid}}$ includes the variables that are needed for the evaluation of the second part of the rule, or for the head. Observe that the first rule (at p) is now local. If the second rule, installed at q, is also local, no further delegations are needed to complete the computation. Otherwise, a new rewriting happens, splitting the rule at q, delegating the second part of the rule as appropriate, and so on.

Observe that an evolution of the state of p may result in installing new rules at q, or in removing some delegations. Deletion of a delegation is captured by updating the predicate guarding the rule. Insertion of a new delegation modifies the program at q. Note that in *Bud* the program of a peer is fixed, and so adding and removing delegations is a novel feature in *WebdamLog*. Implementing this feature requires us to modify the *Bud* program during step 1 of the *WebdamLog* stage.

Finally, we consider relation and peer variables. In all cases presented so far, *WebdamLog* rules could be compiled statically into *Bud* rules. This is no longer possible in the final case. In particular, observe that, if the peer name in an atom in the body of a rule is a variable, then the system cannot tell before the variable is instantiated whether the rule is local or not. In general, we cannot compile a *WebdamLog* rule into *Bud* before all peer and relation variables are instantiated. This is considered in the fourth case.

Relation and peer variables. To illustrate this case, consider a rule of the form

$\mathsf{r_0@p}(\overline{X_0})\text{:-}\ \mathsf{r_1@p}(\$\mathsf{X}),\ \ldots,\$\mathsf{X@p}(\overline{X_i}),\ldots,$

where $r_0@p$ is extensional and $\$\mathsf{X}$ is a variable. This particular rule is relatively simple since, no matter how the variable is instantiated, the rule falls into what we called the basic case. However, it is not a *Bud* rule because of the variable relation name $\$\mathsf{X}$.

Recall that *WebdamLog* rules are evaluated from left to right, and a constraint is that each relation and peer variable must be bound in a previous atom. (This constraint is imposed by the language.) Therefore, when we reach the atom $\$\mathsf{X@p}(\overline{X_i})$, the variable $\$\mathsf{X}$ has been instantiated.

To evaluate this rule, we use two *WebdamLog* stages of the peer. In the first stage, we bind $\$\mathsf{X}$ with values found by instantiating $\mathsf{r_1@p}(\$\mathsf{X})$. Suppose that we find two values for $\$\mathsf{X}$, say t_1 and t_2. We always wait for the next stage to introduce new rules (there are two new rules in this case). More precisely, new rules are introduced during step 1 of the *WebdamLog* computation at

the next stage. In the example, the following rules are added to the *Bud* program at p:
$\mathsf{r}_0@\mathsf{p}(\overline{X_0})$:- $\mathsf{t1}@\mathsf{p}(\overline{X_i}),\ldots,$ $\mathsf{r}_0@\mathsf{p}(\overline{X_0})$:- $\mathsf{t2}@\mathsf{p}(\overline{X_i}),\ldots,$

Even in the absence of delegation, having variable relation and peer names allows the *WebdamLog* engine to produce new rules at run time, possibly leading to the creation of new relations. This is a distinguishing feature of our approach, novel to *WebdamLog* and to our implementation.

This example uses a relation name variable. Peer name variables are treated similarly. Observe that having a peer name variable, and instantiating it to thousands of peer names, allows us to delegate a rule to thousands of peers. This makes distributing computation very easy from the point of view of the user, but also underscores the need for powerful security mechanisms. Developing such mechanisms is in our immediate plans for future work.

5 Experimental Evaluation

Our ongoing experimental evaluation aims to demonstrate that *WebdamLog* programs can be executed efficiently in a highly dynamic distributed setting. We present results demonstrating that rule rewriting and delegation can be implemented efficiently. Other experiments we performed are mentioned at the end of the section. Because of space limitations, they could not be detailed here. They are discussed in [17].

The cost of delegation. In the following experiments, our emphasis is on measuring the *WebdamLog* overhead in dealing with delegations. Recall the steps performed by each peer at each *WebdamLog* stage, described in Section 4. We can break down each step into *WebdamLog*-specific and *Bud*-specific tasks as follows: in the first step(1) inputs are collected: (a)*Bud* reads the input from the network and populates its channels. (b) *WebdamLog* parses the input in channels and updates the dependency graph with new rules. This step is needed for semi-naive evaluation . Second step(2): (a) time is frozen *WebdamLog* and *Bud* invalidate the Δ of each relation that needs to be re-evaluated during fixpoint. Δ are used by the semi-naive evaluation algorithm to optimize its computation. *Bud* invalidates Δ according to instance updates, while *WebdamLog* invalidates Δ according to program updates. (b) *Bud* performs semi-naive fixpoint evaluation for all invalidated relations, taking the last Δ for differentiation. Third step(3): Outputs are obtained (a) *WebdamLog* builds packets of rules and updates to send. (b) *Bud* sends packets.

We report the running time of *WebdamLog* as the sum of steps (1b) and (3a), and the running time of *Bud* as the sum of steps (1a), (2b) and (3b). We report the running time of step (1b) separately, since it involves both *WebdamLog* and *Bud*, and we refer to this step as *mixed*. All running times are expressed as % of the total running time. For each experiment, we show that the running time of *WebdamLog*-specific phases is reasonable compared to the over-all running time.

For the experiments, we use 3 different machines running (i) the Ruby 1.8.7 virtual machine, (ii) the *Bud* system, and (iii) the *WebdamLog* engine described in Section 4. The machines have the following configurations (1) Intel Core i5 M540 @2.53GHz CPU, 4GB RAM, running Ubuntu 12.04; (2) Intel Core 2 Duo U7600 @1.2GHz CPU, 2GB RAM, running Ubuntu 12.04; (3) Intel Xeon 5160 @ 3.00GHz CPU, 1GB RAM, running Mandriva 2010.0.

To ensure that the observed trends are not due to differences in hardware / OS between the peers, we execute each experiment using all possible assignments of peers to machines, for a total of 6 possible assignments. Further, to reduce variability due to noise, we execute each experiment

5 times per peer assignment. Therefore, we executed a total of 30 runs for each experiment. We observed that assignment of peers to machines did not significantly impact running times, and so we report an average measure for each peer.

For the experiments in this section, we use *WebdamLog* rules involving *only extensional relations*, both in the head and in the body. We also support rules with intensional relations in the head and in the body. But for such rules, an essential optimization consists in deriving *only the relevant data and delegated rules*. We intend to conduct experiments with such rules when our system supports optimizations in the style of Magic Set.

Non-local rules. In the first experiment, we evaluate the running time of a non-local rule with an extensional head. Rules of this kind involve delegation. The following rule computes the join of relations $rel1@peer1$ and $rel2@peer2$, and then installs the result, projected on the last column, in $join@peer3$:

```
[at peer1] join@peer3($Z) :- rel1@peer1($X,$Y), rel2@peer2($Y,$Z)
```

Relations $rel1@peer1$ and $rel2@peer2$ each contain 1000 tuples that are pairs of integers, with values drawn uniformly at random from the 1 to 100 range. In the next table, we report the total running time of the program at each peer, as well as the break-down of the time into *Bud*, *WebdamLog*, and *mixed*.

The portion of the over-all time spent on *WebdamLog* computation on peer1 is fairly high at 7.9%. However, this is because the total processing time at peer1 is low, and much of the processing is spent on rule rewriting, to delegate the join to peer2. The portion of the over-all time spent on *WebdamLog* computation on peer2 is lower, because this peer spends more time computing the join. Peer peer3 spends a significant portion of the time in the *mixed* step, because the bulk of this peer's work involves installing a new rule, to be populated by the result of the join.

Non local rules

	WDL	*Bud*	*mixed*	total
peer1	7.9%	57.3%	34.7%	0.18s
peer2	6.7%	73.2%	20.0%	0.95s
peer3	0.2%	65.0%	34.7%	0.10s

Relation and peer variables

	WDL	*Bud*	*mixed*	total
p	12.9%	77.2%	9.7%	1.28s
remote1	1.1%	60.3%	38.5%	0.09s
remote2	1.3%	58.7%	40.0%	0.07s

Relation and peer variables. In the second experiment, we evaluate the execution time of a *WebdamLog* program for the distributed computation of a union. The following rule uses relation and peer variables and executes at peer p:

```
[at p] union@p($X) :- peers@p($Y,$Z), $Y@$Z($X)
```

We assume that there exists an extensional relation $peers@p$, with each tuple corresponding to a valid combination of peer name and relation name. In our experiment, $peers@p$ contains 12 tuples and 3 distinct peers, i.e., the rule computes the union of 12 relations, 4 per peer. Each relation participating in the union contains 500 tuples, each with a single integer column, and with values for the attribute drawn independently at random from the 1 to 100 range. We record the performance of p, where the union is computed, and of the other two peers, remote1 and remote2, that are sending their entire relations to p.

WebdamLog overhead peaks at around 13% on p, since this peer has to dynamically create and install 12 new rules. Nonetheless, this overhead is manageable. As expected, the running time on

both remote peers is low, and the *WebdamLog* portion of the computation is insignificant.

This section was only meant as an illustration of the on-going experiments we are conducting. For instance, we are also performing the following experiments [17]: (i) We compare the querying and maintenance costs of materialized views to that of delegation-supported views; and (ii) We evaluate the gains achieved by using provenance graphs to support deletions of facts and rules.

6 Conclusion

This paper presents an implementation of the *WebdamLog* language introduced in [4]. We demonstrate the feasibility of an approach based on *WebdamLog* to support exchanges of data and rules between peers that evolve very rapidly in a distributed and dynamic environment. To benefit from previous datalog optimization and efficient network communication, we rely on *Bud* to support the basic functionality of distributed datalog. We show how to realize the higher level features of *WebdamLog*, notably delegation, on top of *Bud*, using logical rule rewriting.

The use of declarative languages, in particular datalog extensions, for distributed data management has been advocated in [2, 5, 13]. Several systems have been developed based on the declarative paradigm [12, 15, 14], with performance comparable to that of systems based on imperative languages. Our implementation uses the *Bud* system [9]. The language Dedalus [7] has been proposed as a formal foundation for *Bud*. We prefer here to use the language *WebdamLog*, in particular because it features delegation.

In the future we expect to demonstrate the use of our system with a much larger number of peers. As our ongoing research, we are exploring how various datalog performance optimization techniques, such as query-subquery, may be extended and applied to our setting.

References

[1] S. Abiteboul. Managing an XML warehouse in a P2P context. In *CAiSE*, 2003.

[2] S. Abiteboul, O. Benjelloun, and T. Milo. The Active XML project: an overview. *VLDB J.*, 2008.

[3] S. Abiteboul et al. A model for web information management with access control. In *WebDB*, 2011.

[4] S. Abiteboul et al. A rule-based language for Web data management. In *PODS*, 2011.

[5] S. Abiteboul and N. Polyzotis. The data ring: Community content sharing. In *CIDR*, 2007.

[6] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *JCSS*, 1991.

[7] P. Alvaro and et al. Dedalus: Datalog in Time and Space. Technical report, UC Berkeley, 2009.

[8] É. Antoine et al. [Demo] Social Networking on top of the WebdamExchange System. In *ICDE*, 2011.

[9] Berkeley Orders Of Magnitude. Bloom programming language. `http://www.bloom-lang.net/`.

[10] M. J. Franklin et al. From databases to dataspaces management. *SIGMOD Record*, 2005.

[11] A. V. Gelder. Negation as failure using tight derivations for general logic programs. *JLP*, 1989.

[12] S. Grumbach et al. Netlog, a rule-based language for distributed programming. In *PADL*, 2010.

[13] J. M. Hellerstein. The declarative imperative. *SIGMOD Record*, 2010.

[14] B. T. Loo. Declarative networking: language, execution and optimization. In *SIGMOD*, 2006.

[15] B. T. Loo et al. Implementing declarative overlays. In *SOSP*, 2005.

[16] A. Peter et al. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR*, 2011.

[17] WebdamLog engine. Experiments. `http://webdam.inria.fr/webdamlog/experiments`.